UNIVERSITÄT AUGSBURG
Universitätsbibliothek

OPEN ACCESS
OPUS AUGSBURG

Formal Semantics of Asbru - V2.12

Michael Balser, Christoph Duelli, Wolfgang Reif, Jonathan Schmitt

Angaben zur Veröffentlichung / Publication details:

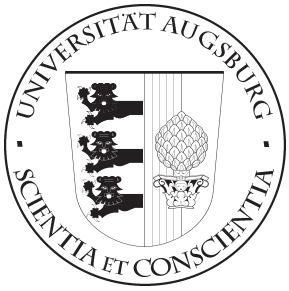
Balser, Michael, Christoph Duelli, Wolfgang Reif, and Jonathan Schmitt. 2006. “Formal Semantics of Asbru - V2.12.” Augsburg: Universität Augsburg.

Nutzungsbedingungen / Terms of use:

licgercopyright

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under these conditions:
Deutsches Urheberrecht
Weitere Informationen finden Sie unter: / For more information see:
https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publiz/

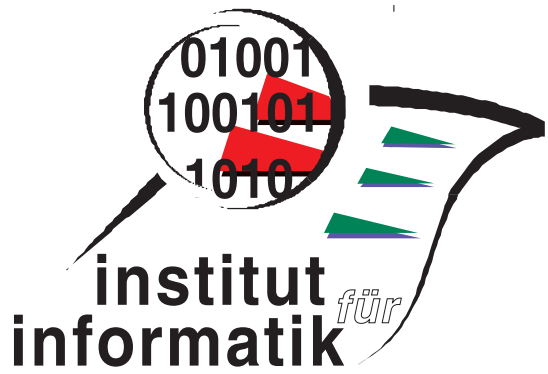
UNIVERSITÄT AUGSBURG



Formal Semantics of Asbru – V2.12

M. Balser, C. Duelli, W. Reif, J. Schmitt

Report 2006-15 Juni 2006



INSTITUT FÜR INFORMATIK
D-86135 AUGSBURG

Copyright © M. Balser, C. Duelli, W. Reif, J. Schmitt
Institut für Informatik
Universität Augsburg
D–86135 Augsburg, Germany
http://www.Informatik.Uni-Augsburg.DE
— all rights reserved —

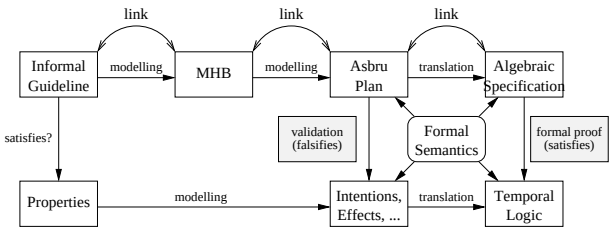
Abstract

This paper gives part of the formal semantics of a planning language called Asbru which has been specifically designed for the medical framework. A formal semantics is an important step within the Protocure project which is concerned with the quality assurance of medical guidelines and protocols. We have constructed a formal semantics in the style of statecharts, which leads to a compact, graphical overview of the operational behaviour. In this style, the semantics documents the language best.[1]

This paper is a revised and extended version of [1].

1 Introduction

This work is part of a European project called Protocure [10], which is concerned with the quality assurance of medical protocols. The idea is to model existing informal medical guidelines and protocols in the planning language Asbru [7] [11] and to verify certain properties. Already, Asbru has been used to formalize a variety of examples from different fields of medicine: diabetes mellitus, jaundice in new born babies, artificial ventilation of prematured babies, treatment of breast cancer and others. Other approaches to model medical protocols are e.g. [3] [8] [6]. One of our major goals is to further utilize formal methods in the medical domain by verifying properties of protocols with mathematical rigour by automatic and interactive verification methods, leading to the following overall picture.



Defining a formal semantics of Asbru is an important step within this project. It is the basis for applying formal methods and also for validating Asbru plans by simulation. On the other hand it should also help to understand the Asbru language with all its details.

Our semantics for Asbru is presented as a set of statecharts. The notation has been very useful for further discussions within our heterogenous group of people from formal methods, medicine, planning, knowledge bases and language design.

For the statecharts to be a base for a formal language definition, it is necessary to define a formal semantic for them, too. The semantics of the statecharts of this paper have been defined by W. Damm in [4].

The paper is organized as follows. Section 2 summarises the changes compared to [1]. In Sect. 3 we will give a short overview of Asbru followed by notational issues in Sect. 4. Section 5 gives an overview of the semantics. The hierarchy of plans is explaind in Sect. 6, which is followed by the basic plan state model of Asbru in Sect. 7. This model is enriched with further important concepts of Asbru in Sections 8 to 14. Section 15 gives an outlook on how the interactive theorem prover KIV is used to formally verify properties of Asbru plans and Sect. 16 concludes.

2 Revision History

A first version of the Asbru semantics has been published in [1]. This version has been revised and extended as follows.

- SOS rules have been removed as state charts sufficiently communicate the formal semantics for our working group. A list of SOS rules can be derived from the given state charts.
- New body types ‘on abort’, ‘on suspend’, ‘on abort on suspend’, and ’if then else’ have been added.
- The formal semantics for cyclical plans has been extended.
- Flags ‘overridable’ and ‘manual’ have been added to conditions. These flags replace the original ‘activate-mode’.

[1]This work has been partially funded by the European Commission’s IST program, under contract number IST-FP6-508794 Protocure II.

plan Regular_Treatments_3
  intentions
    intermediate-state maintain *bilirubin* $\neq$ transfusion
    overall-state achieve *bilirubin* $=$ observation
  conditions
    filter-precondition *bilirubin* $\neq$ transfusion
    abort-condition *bilirubin* $=$ transfusion
      $\vee$ *bilirubin* $=$ pt-intensive
        $\wedge$ *bilirubin_decrease* < 1
          $[[-2\text{ h}, _], [_, 0\text{ h}], [_, _], \mathit{now}]$
  plan-body any-order
    wait-for Observation
    plan-activation Phototherapy_Intensive
    plan-activation Phototherapy_Normal_Prescription
    plan-activation Phototherapy_Normal_Recommended
    plan-activation Observation

Figure 1: Example Asbru plan from Jaundice case study

- Setup, suspend and reactive conditions have been added.
- The evaluation of conditions has been revised. Predicates 'satisfied' and 'satisfiable' have been defined to properly formalize the semantics of time annotated conditions.
- A first discussion of effects has been added.
- Minor errors have been corrected.

3 Asbru in a Nutshell

As an example, Fig. 1 displays a simplified version of one of the plans in the jaundice case study. Treating jaundice in newborn babies requires monitoring the level of bilirubin in the blood. Quantitative bilirubin levels are abstracted to qualitative values observation, pt_normal, pt_recommended, pt_intensive, and transfusion. The intention of this treatment plan is to maintain a bilirubin level lower than transfusion and to finally achieve a very low level of bilirubin called observation. The filter condition states that this plan is only applicable, if the bilirubin level is not too high in the beginning. The plan will be aborted, if bilirubin is too high or if it is very high and the decrease within the last two hours was not large enough. Four different alternative treatments are available. The applicability of these alternatives is determined by their own filter conditions (which are not contained in Fig. 1). For example, plan Phototherapy_Intensive can only be used, if bilirubin level is pt_intensive. Because of the "wait-for" construct, the successful completion of plan Observation is mandatory, other plans are optional.

Asbru is a plan oriented language. Several plans are organized in a hierarchy of plans. A parent plan can refer to other sub plans in its plan body. Conditions are used to control the applicability of a plan and to monitor its execution. Conditions can be monitored over time according to so called time annotations. The sub plans in the plan body can be organized using different body types (e.g. any-order). The current state of a plan – especially if a plan has been rejected, aborted, or completed – is propagated according to the plan hierarchy to its parent and sub plans. If a plan is mandatory, it must be completed for its superplan to complete, otherwise it may also be rejected or aborted.

4 Notation

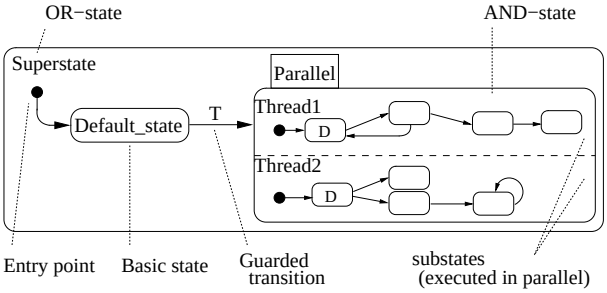
4.1 EBNF

We will use an EBNF-like notation to describe the syntax of constructs of Asbru. Terminal symbols are written in normal style, names of the grammar rules are typeset in *italic*. Square brackets $[\,\cdot\,]$ denote optional parts, and alternatives are written as $(\,\cdot \mid \cdot\,)$. Zero or more repetitions are denoted as $\cdot\,^*$.

4.2 Statecharts Notation

A statechart is a directed graph representing a state machine (a nondeterministic automaton). It is used to specify a system's dynamic behaviour. In this paper we will adopt the syntax of STATEMATE [9]. We will explain its basic features and semantics on the basis of Figure 2.

States are depicted as rounded rectangles. *Superstate* con-



T : *event* [*condition*] /*action*

Figure 2: Statechart notation

tains two substates. As the system can be only in one of these at a given time, *Superstate* is called an OR-state. When the system enters *Superstate*, it's initially in both *Superstate* itself and its substate *Default_state*. The default substate is marked with an arc pointing from a black bullet to the state.

Possible state transitions are represented as directed arcs that may be labelled with guards of the form *event* [*condition*] /*action*. A guarded transition can only be taken when *event* occurs and *condition* holds at the same time. Note that an enabled transition *must* be taken (nondeterministic choice if several transitions from a state are enabled).

Parallel is an AND-State. Its substates *Thread1* and *Thread2* – separated by a dashed line – are executed synchronously in parallel. Once transition T is enabled, *action* will be executed and then *Parallel* will become active, changing the system's active states to *Superstate*, *Parallel*, *Thread1*, *Thread2*, *Thread1.D* and *Thread2.D*. By means of composite AND- and OR-states, we can create a *state hierarchy*, thus facilitating the readability of the statechart.

4.3 Statecharts Interpretation

The state charts are formally interpreted according to the semantics of [4]. The semantics of state charts is complex in general. However, interpretation of the given state charts in this paper are very intuitive.

5 Semantics Overview

Plans may refer to sub plans in their plan body leading to a hierarchy of plans as described in Sect. 6. The behaviour of a single plan is defined in the so called plan state model: conditions are used to control selection and execution of plans. This is explained in Sect. 7. The relationship between parent and sub plans is encoded in events which synchronize the execution of sub plans (see Sect. 8), and the concept of propagation (see Sect. 9). The definition of mandatory and optional sub plans is discussed in Sect. 10. The special case of cyclical execution of sub plans is explained in Sect. 11. Conditions are evaluated by an underlying data abstraction unit (see Sect. 13). The abstraction unit also takes care of monitoring data over a longer period of time as defined by time annotations in conditions and manipulating data as described by effects. In our semantics intentions describe properties of plans and can be used as proof obligations for verification (see Sect. 15). Effects are wanted or unwanted side effects of plans and change the state of the abstracted patient (see Sect. 14).

So far, we only consider part of Asbru version 7.2 (as described in [11]) within this paper. However, we claim that the major concepts of Asbru are covered. Concepts which are neglected, include

- local variables and return values,
- context of parameters,
- more complex cyclical plan execution.

Either these topics are well understood (e.g. local variables) or they are not used in our case studies (e.g. parameter context). The formal semantics of cyclical plans is still work in progress.

It is necessary to distinguish between a data structure representing the patient and the known data about the patient. Treatments affect the patient, while only measurements can make the results of a treatment - or, more generally, the state of the patient - visible. It is self evident, that conditions may only be evaluated over the known measurements, not the status of the patient itself.

As Asbru plans may refer to important time points of other asbru plans in the past, it is necessary to write down

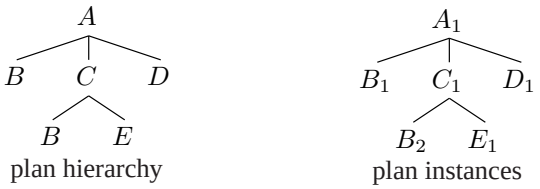


Figure 3: Plan hierarchy

a history of plan state changes. As conditions might set past measurements in relation, it is also necessary to log the history of measurements.

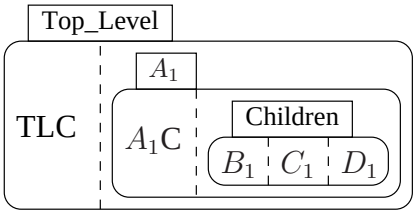
6 Plan Hierarchy

In Asbru, plans are organized in a hierarchy as shown on the left of Figure 3: a parent plan A refers to a number of sub plans B, C, and D in its plan body. Sub plans may refer to further plans resulting in a tree hierarchy. The plan name is used to reference a plan.

One and the same plan may occur several times within this hierarchy (e.g. plan B). Therefore we distinguish between plan references and plan instances. Each reference corresponds to a unique instance. On the right of Fig. 3, plan references have been numbered to give unique instances. The first occurence of plan B is instance B_1, the second is instance B_2.

6.1 Semantics overview

In our semantics, all existing instances of plans are executed in parallel. The hierarchy of instances is preserved. That is, every plan, that has children, directly controls the execution of its children. Additionally one top level control is launched for the main plan. For the situation in Fig. 3, we denote this top level control with the following statechart.



Further subplans of plans B_1, C_1 and D_1 are ommited, for the figure to be better readable.

7 Plan State Model

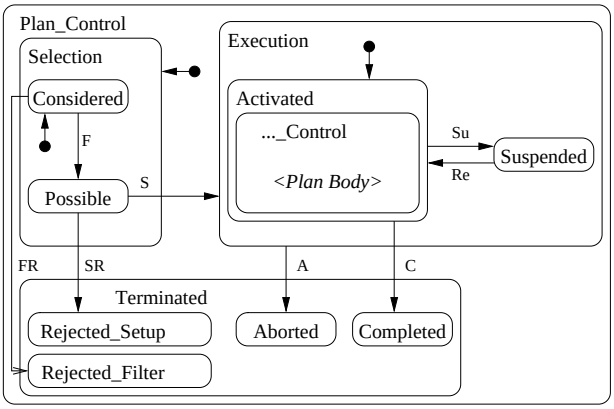
The overall plan state model defines the semantics of the different conditions of a plan. Conditions are used to decide if the plan body is applicable (selection phase) and while executing the body, if execution should be interrupted (execution phase).

7.1 Syntax

The syntax of a plan is as follows.

plan = plan *name*
[*intentions*]
[*effects*]
[conditions
[filter-precondition *temporal-pattern bool bool*]
[setup-precondition *temporal-pattern bool bool*]
[suspend-condition *temporal-pattern bool bool*]
[reactivate-condition *temporal-pattern bool bool*]
[abort-condition *temporal-pattern bool bool*]
[complete-condition *temporal-pattern bool bool*]]
plan-body

A plan consists of intentions (see Sect. 15), effects (see Sect. 14) the definition of conditions (see below), and the plan body (see Sect. 8). The different conditions consist of a temporal pattern and two boolean flags stating whether the condition is *overridable* or can be *manual*ly triggered.



S : [*satisfied(setup_cond)*]
F : [*satisfied(filter_cond)*]
FR : [*not satisfied(filter_cond)*]
SR : [*not satisfiable(setup_cond)*]
Su : [*satisfied(suspend_cond)*]
Re : [*satisfied(reactivate_cond)*]
A : [*satisfied(abort_cond)*]
C : [*satisfied(complete_cond)*]

Figure 4: Semantics of plan state model

7.2 Semantics overview

A variation of the standard plan state model described in [7] is given in Fig. 4 to define the semantics of conditions. The *Plan_Control* is divided into the selection phase *Selection* and the execution phase *Execution*. Initially a plan is *Considered*. In this state, the filter condition *filter_cond* is checked. If this condition is satisfied, control advances to state *Possible* (transition F). If the filter condition is not satisfiable the state is changed to *Rejected_Filter*, else the state remains at state *Considered*. In state *Possible*, the setup condition is evaluated. If the setup condition is satisfied, control advances to the execution phase. Otherwise, the plan is not immediately rejected. Only, if the setup condition is not satisfiable anymore, the plan is rejected. (For details on the definition of satisfied and satisfiable, see below.) In state *Activated*, the sub plans of the current plan are executed, if the plan is not user performed or an ask plan. This is described in Sect. 8. The execution can be either completed successfully (transition C) or aborted in the case of emergency patient readings (transition A). If the suspend condition is satisfied or the super plan suspends, the state changes to *Suspended*. While in this state, the plan no longer starts further subplans. Execution of already started subplans is also suspended if they are acticvated and proceeds to the next synchronisation point if the subplan is currently in selection phase. Plans in state *Suspended* evaluate the abort condition, the complete condition is not evaluated. The state *Suspended* can be left, once the reactivate condition is satisfied and the parent is activated, in which case the plan state changes back to state *Activated*. We refer to *Terminated*, if the reason for termination – rejection, completion, or abortion – is irrelevant.

For evaluating conditions, two additional flags *overridable* and *manual* have to be taken into account. If the first flag *overridable* is true, an external signal *override* can immediately trigger the condition. If the second flag *manual* is true, an external signal *manual* is necessary to acknowledge the condition.

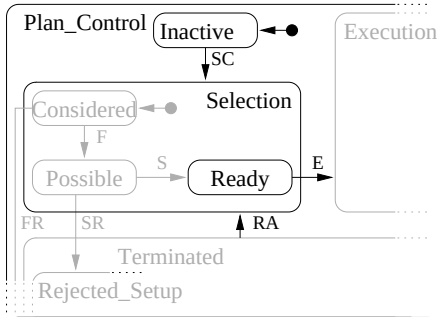
For a given plan C, the formal semantics of the setup condition is as follows:

satisfied(setup_cond) =
satisfied(setup_tp)

$\wedge$ ($\neg$ setup_manual $\vee$ C.setup_manual)
$\vee$ setup_overridable $\wedge$ C.setup_override

satisfiable(setup_cond) =
satisfiable(setup_tp)
$\vee$ setup_overridable

satisfied(setup_tp) and satisfiable(setup_tp) are defined in Section 13. The semantics of the other conditions is analogous.



SC : *consider*
E : *activate*
RA : *retry*

Figure 5: Synchronization states in plan state model

8 Plan Body

How the parent plan controls the plan instances in its plan body will be explained next.

8.1 Syntax

The syntax of the plan body is as follows.

plan-body
= plan-body (sequential|parallel|any-order|unordered
|on abort |on suspend|on abort/suspend
|if then else(conditional)|ask(parameter))
|cyclical
[wait-for-optional] [retry-aborted]
wait-for
(plan-activation *name*)*

The type of the body is either ‘sequential’, ‘parallel’, ‘any order’, ‘unordered’, ‘on abort’, ‘on suspend’, ‘on abort/suspend’, ‘if then else’, or ‘ask’. Additionally the body can be cyclical (see Sect. 11). With option ‘retry-aborted’ aborted sub plans will be retried. The ‘wait-for’ construct defines mandatory and optional plans (see Sect. 10 – here also the option ‘wait-for-optional’ is explained), and the names of the sub plans are listed as plan-activations.

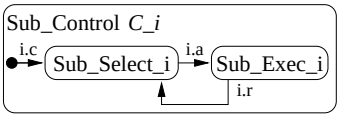
8.2 Semantics overview

Sub plans $C_1, \ldots, C_n$ are controlled in the body of a plan P. Their execution can be organized differently: they can be executed *sequentially* starting with C_1, they can be executed in parallel either with synchronization (*parallel*) or without synchronization (*unordered*) of the selection and execution phases, and finally they can be executed sequentially, but *any order*, i.e., only one sub plan is executed at once, but the sequence is not fixed.[2]. Aborted plans can be retried. Additionally, a sub plan can be executed depending on a condition or on the the abortion or suspension of another plan.

In order to allow synchronization of the selection and execution phases of the sub plans, and the retrial of plans, the plan state model has to be enriched with intermediate states *Inactive*, *Ready*, and additional transitions SC, E, RA, resulting in the adapted statechart of Fig. 5. The additional events *consider*, *activate*, and *retry* are used to externally control progress of a plan. A parent plan can thus synchronize the sub plans in its plan body. For this, the *Activated* state of the parent is refined with a controlling statechart.

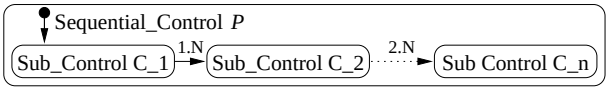
If no restriction on the progress of sub plan C_i is required, the controlling statechart *Sub_Control_i* C_i of Fig. 6 can

[2] Cyclical execution will be explained in Sect. 11



$i.c$: $/C_i$.*consider*
$i.a$: [**in**(C_i.*Ready*)] $/C_i$.*activate*
$i.r$: [*retry-aborted* $\equiv$ yes $\wedge$ **in**(C_i.*Aborted*)] $/C_i$.*retry*

Figure 6: Controller for executing a sub plan with no restrictions



$$i.N : \begin{bmatrix} \textit{retry-aborted} \equiv \text{no} \wedge \textbf{in}(C_i.\textit{Terminated}) \\ \vee\ \textit{retry-aborted} \equiv \text{yes} \wedge \textbf{in}(C_i.\textit{Completed}) \\ \vee\ \textit{retry-aborted} \equiv \text{yes} \wedge \textbf{in}(C_i.\textit{Rejected}) \end{bmatrix}$$

Figure 7: Controller for sequential execution

be used. Sub plan C_i is considered immediately (transition $i.c$) and is activated as soon as it reaches state *Ready* (transition $i.a$). If option "retry-aborted" is chosen, then the sub plan is retried, if it aborts (transition $i.r$).

The different body types may oppose restrictions on the execution of sub plans. This is done by deferring the generation of the newly added events. Controlling statecharts for the different body types are explained next.

8.2.1 Sequential execution

The controller of Fig. 7 considers the first sub plan. As soon as it terminates, we continue with the second plan (transition $1.N$). During execution of one plan, no synchronization is required. Thus, we use *Sub_Control* C_i to execute each sub plan. In the case of "retry-aborted", a sub plan is considered to terminate, if it is either completed or rejected. If it aborts, the controller *Sub_Control* C_i will take care of retrying the plan immediately (see Fig. 6).

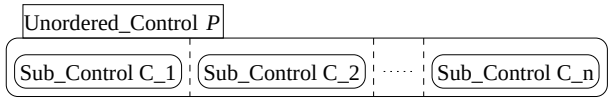
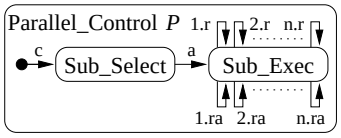


Figure 8: Controller for unordered execution



c : $/C_1$.*consider*; ...; C_n.*consider*
a : [$\bigwedge_{i=1}^{n}$ **in**(C_i.*Ready*) $\vee$ **in**(C_i.*Rejected*)]
/ C_1.*activate*; ...; C_n.*activate*
$i.r$: [*retry-aborted* $\equiv$ yes $\wedge$ **in**(C_i.*Aborted*)]
/ C_i.*retry*;
$i.ra$: [*retry-aborted* $\equiv$ yes $\wedge$ **in**(C_i.*Ready*)]
/ C_i.*activate*;

Figure 9: Controller for parallel execution

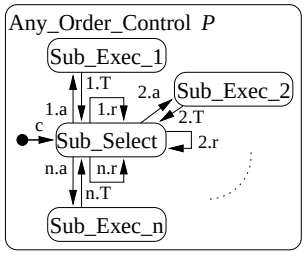
8.2.2 Unordered execution

The controller in Fig. 8 executes the sub plans in parallel and no further synchronization is necessary.

8.2.3 Parallel execution

The parallel operator (see Fig. 9) synchronizes selection and execution phases of all sub plans. The sub plans are considered immediately (transition c). They may only proceed to *Activated* state, if all sub plans are *Ready* (transition a).[3]

If the retry flag is set, all plans that abort are immediately retried. They are reactivated as soon as they again reach state *Ready*.



$$i.a: \quad \left[\begin{array}{l}(\ \bigwedge_{k=1}^{n} \neg\ \textbf{in}(C_k.\textit{Aborted}) \\ \quad \vee\ \textit{retry-aborted} \equiv \text{no}) \\ \wedge\ \textbf{in}(C_i.\textit{Ready})\end{array}\right]$$
$$/\quad C_i.\textit{activate}$$
$$i.T: \quad [\textbf{in}(C_i.\textit{Terminated})]$$
$$i.r: \quad [\textit{retry-aborted} \equiv \text{yes} \wedge \textbf{in}(C_i.\textit{Aborted})]$$
$$/\quad C_i.\textit{consider};$$

Figure 10: Controller for any order execution

8.2.4 Any order execution

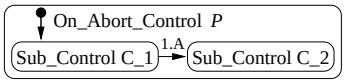
In Fig. 10 only the selection phases are started in parallel. The execution phases of the sub plans are synchronized such that at most one sub plan is active at the same time. For this the plans are considered immediately (transition c). The first plan to become selectable is activated (transition $i.a$). Only if this plan terminates (transition $i.T$) another one can be activated. If several sub plans reach state *Ready* simultaneously, the choice is nondeterministic. If option "retry-aborted" is chosen, then transition $i.r$ is used to initiate a retrial.

As long as any plan is set to aborted (and retry-aborted is set), no other plan can reach the activated state. Therefore it is guaranteed, that every aborting plan gets to reenter the selection phase.

8.2.5 On abort execution

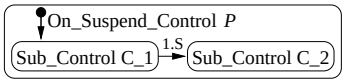
This type of body contains exactly two sub plans C_1 and C_2. It executes sub plan C_1 using standard control. If the plan aborts, then C_2 is executed (see transition $1.A$ in Fig. 11). Option "retry-aborted" only affects execution of C_2. If C_1 aborts, then transition $1.A$ overrides transition $1.r$ of controller *Sub_Control* C_1 (see Fig. 6) which would have initiated a retrial of C_1.

[3] As will be explained in Sect. 10, it is sufficient that all mandatory sub plans are *Ready* in order to proceed to *Activated* state.



$$1.A: \quad [\textbf{in}(C_1.\textit{Aborted})]$$

Figure 11: Controller for on abort execution



$$1.S: \quad [\textbf{in}(C_1.\textit{Suspended})]$$

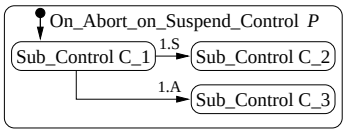
Figure 12: Controller for on suspend execution

8.2.6 On suspend execution

This type of body executes sub plan C_1 using standard control. If the plan suspends, then C_2 is executed (see transition $1.S$ in Fig. 12). Option "retry-aborted" only affects execution of C_2. Execution of C_1 is not affected by the start of C_2.

8.2.7 On abort on suspend execution

This type of body executes sub plan C_1 using standard control. If the plan suspends, then C_2 is executed (see



$$1.S: \quad [\textbf{in}(C_1.\textit{Suspended})]$$
$$1.A: \quad [\textbf{in}(C_1.\textit{Aborted})]$$

Figure 13: Controller for on suspend execution

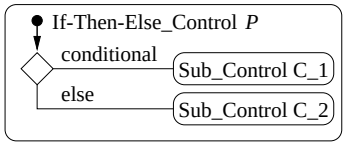


Figure 14: Controller for on suspend execution

transition $1.S$ in Fig. 13). Option ”retry-aborted” affects both execution of C_2 and C_1. Execution of C_1 is not affected by the start of C_2. If C_1 aborts, C_3 is started. The start of C_3 and C_2 are mutual exclusive.

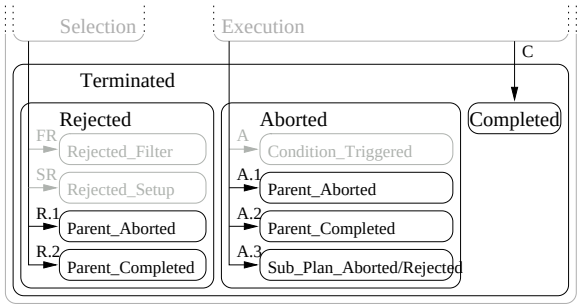
8.2.8 If then else execution

This type of body executes sub plan C_1, if the conditional evaluates to true. If not, sub plan C_2 will be executed. (See Figure 14.) If no plan C_2 is present and the conditional evaluates to false, plan P will be completed immediately.

9 Propagation

The parent is able to control and synchronize progress of its sub plans as described in the previous section. Nevertheless additional control to propagate execution states of a sub plan to its parent and vice versa is necessary. For example, if a (mandatory) sub plan C_i aborts, then also the parent aborts. This is known as propagation in Asbru. There are a number of dependencies between sub plans and parent similar to this example. All of them are displayed as additional or refined transitions in Fig. 15.

If it is relevant, it can be further distinguished between the states *Parent_Aborted*, if the superplan of the current plan aborts, *Parent_Completed*, if the superplan completes, *Child_Aborted*, if a crucial subplan aborts or *Child_Completed*, if certain subplans complete. This can be relevant, if the completion of a subplan denotes the occurance of an unwanted event, e.g. the subplan completes upon detection of critical blood pressure, but the superplan only deals with elevated (but non-critical) blood pressure.



$$R.1 \quad : [\textbf{in}(P.Aborted)]$$
$$R.2 \quad : [\textbf{in}(P.Completed)]$$

$$A.1 \quad : [\textbf{in}(P.Aborted)]$$
$$A.2 \quad : [\textbf{in}(P.Completed)]$$
$$A.3 \quad : \left[\bigvee_{i=1}^{n} \begin{pmatrix} \textbf{in}(C_i.Rejected) \\ \vee\ \textbf{in}(C_i.Aborted) \end{pmatrix}\right]$$

$$C \quad : \begin{bmatrix} satisfied(complete_cond) \\ \wedge \bigwedge_{i=1}^{n} \textbf{in}(C_i.Completed)) \end{bmatrix}$$

Figure 15: Semantics of propagation

10 Continuation Specification

Some of the sub plans are mandatory for the successful execution of the parent plan, others are optional. The ”wait-for” construct determines, which or how many of the sub plans the parent requires to complete successfully.

10.1 Syntax

wait-for
= (*abstract-wait-for* |all|one|*number* [, list]|none)

The parent either requires all, one, a fixed number or none of its sub plans to complete successfully. Alternatively, a logical expression *abstract-wait-for* can be used to specify the set of plans to complete successfully. The syntax for the logical expression is as follows.

abstract-wait-for
= *name*

| not *abstract-wait-for*
| *abstract-wait-for* (and|or|xor) *abstract-wait-for*

We will also define the option ”wait-for-optional” of the plan body (see Sect. 8) in this section.

10.2 Semantics Overview

Let $C_1, \ldots, C_n$ be the sub plans of plan P. The ”wait-for” construct of P is transformed into a formula with variables $v_1, \ldots, v_n$. For example the construct

$$\textit{wait-for } (C_1 \text{ and } C_2) \text{ or } C_3$$

(which either requires plans C_1 and C_2 or C_3 to complete) is transformed into the following formula.

$$(v_1 \wedge v_2) \vee v_3$$

Within this formula, the variables can be replaced by boolean conditions concerning the current state of each sub plan. For our example, we can determine, if enough sub plans have completed already, by replacing the variables v_i with the conditions **in**(C_i.*Completed*) leading to the expression

$$\begin{aligned}&(\textbf{in}(C_1.\textit{Completed}) \wedge \textbf{in}(C_2.\textit{Completed}))\\ \vee\ &\textbf{in}(C_3.\textit{Completed})\end{aligned}$$

If and only if this expression evaluates to *true*, enough sub plans have completed.

Similarly we can determine, if too many sub plans have been rejected or aborted, such that the parent cannot complete successfully any more. By replacing variables v_i with conditions

$$\neg\ (\textbf{in}(C_i.\textit{Rejected}) \vee \textbf{in}(C_i.\textit{Aborted}))$$

we receive

$$\begin{aligned}&\neg\ (\textbf{in}(C_1.\textit{Rejected}) \vee \textbf{in}(C_i.\textit{Aborted}))\\ \wedge\ &\neg\ (\textbf{in}(C_2.\textit{Rejected}) \vee \textbf{in}(C_2.\textit{Aborted}))\\ \vee\ &\neg\ (\textbf{in}(C_3.\textit{Rejected}) \vee \textbf{in}(C_3.\textit{Aborted}))\end{aligned}$$

This expression evaluates to *true*, if and only if still enough sub plans can complete. Vice versa, if it evaluates to *false*, too many sub plans have aborted and the parent needs to abort also.

Summarized, the following steps are necessary to take care of the continuation specification of P.

1. Extract the ”wait for” construct of P,
2. turn it into a formula with variables $v_1, \ldots, v_n$,
3. replace the variables with a given list of conditions $[b_1, \ldots, b_n]$,
4. evaluate the resulting expression.

We will use a function $\text{cs}_P([b_1, \ldots, b_n])$ to abstract from these steps.

With this function, the transitions $A.3$ and C of Sect. 9 can be adapted as follows.

$$\begin{aligned}A.3 \quad &: \left[\neg\ \text{cs}_P\left(\left[\neg\ \begin{pmatrix}\textbf{in}(C_i.\textit{Rejected})\\ \vee\ \textbf{in}(C_i.\textit{Aborted})\end{pmatrix}\right]\right)\right]\\ C \quad &: \begin{bmatrix}\textit{satisfied(complete_cond)}\\ \wedge\ \text{cs}_P([\textbf{in}(C_i.\textit{Completed})])\end{bmatrix}\end{aligned}$$

The parent plan will abort, if too many plans have been rejected or aborted (transition $A.3$), it will complete, if enough sub plans have completed (transition C).

Also – for parallel execution (see Sect. 8.2.3) – we will start execution, if enough sub plans are selected. For this, we adapt transition a as follows.

$$\begin{aligned}a: \quad &[\text{cs}_P([\textbf{in}(C_i.\textit{Selected})])]\\ /\quad &C_1.\textit{activate}; \ldots; C_n.\textit{activate}\end{aligned}$$

If the body contains option ”wait-for-optional”, then again enough sub plans must complete to satisfy the continuation specification, and the parent plan additionally waits for all sub plans to at least terminate (either with or without success). Therefore transition C needs to be refined further.

$$C \quad : \begin{bmatrix}\textit{satisfied(complete_cond)}\\ \wedge\ \text{cs}_P([\textbf{in}(C_i.\textit{Completed})])\\ \wedge\ (\quad \textit{wait-for-optional} \equiv \text{yes}\\ \quad \rightarrow \bigwedge_{i=1}^{n} \textbf{in}(C_i.\textit{Terminated}))\end{bmatrix}$$

10.3 Semantics

Let P be a plan with sub plans $C_1, \ldots, C_n$. Two functions wf and awf are used to turn the ”wait for” construct *wf* into a formula $\varphi \in \mathbf{F}$.

$$\begin{aligned}\text{wf} \quad &: \textit{wait-for} \rightarrow \mathbf{F}\\ \text{awf} \quad &: \textit{abstract-wait-for} \rightarrow \mathbf{F}\end{aligned}$$

Function wf translates the special cases – i.e. all, one, a fixed number or none of the sub plans must complete – into a formula. A more general specification of type *abstract-wait-for* is taken care of in function awf.

Function wf is defined as follows.

$$\begin{aligned}
\text{wf(all)} &= \bigwedge_{i=1}^{n} v_i \\
\text{wf(one)} &= \bigvee_{i=1}^{n} v_i \\
\text{wf}(m, list) &= \bigvee_{(p)_i \in \text{perm}([1,\ldots,n])} \bigwedge_{i=1}^{m} v_{p_i} \wedge p_i \in list \\
\text{wf(none)} &= \textit{true} \\
\text{wf}(\textit{awf}) &= \text{awf}(\textit{awf})
\end{aligned}$$

If a fixed number m of sub plans are required to complete, a formula is generated which takes all permutations perm$([1,\ldots,n])$ of numbers $1,\ldots,n$ and requires the first m sub plans of the permutated list to complete. For the semantics it does not matter that the resulting formula is highly redundant.

The definition of function awf is straightforward.

$$\begin{aligned}
\text{awf}(\textit{name}) &= v_i, \quad \text{if } \textit{name} \equiv C_i \\
\text{awf}(\text{not } \textit{awf}) &= \neg\, \text{awf}(\textit{awf}) \\
\text{awf}(\textit{awf}_1 \text{ and } \textit{awf}_2) &= \text{awf}(\textit{awf}_1) \wedge \text{awf}(\textit{awf}_2) \\
\text{awf}(\textit{awf}_1 \text{ or } \textit{awf}_2) &= \text{awf}(\textit{awf}_1) \vee \text{awf}(\textit{awf}_2) \\
\text{awf}(\textit{awf}_1 \text{ xor } \textit{awf}_2) &= \text{awf}(\textit{awf}_1) \wedge \neg\, \text{awf}(\textit{awf}_2) \\
&\quad \vee \neg\, \text{awf}(\textit{awf}_1) \wedge \text{awf}(\textit{awf}_2)
\end{aligned}$$

Using these functions, we can define a function

$$\text{cs}_P : [\text{bool}] \rightarrow \text{bool}$$

which takes a list of boolean values and evaluates for a plan P the continuation specification wf_P by substituting all variables $v_1,\ldots,v_n$ in the generated formula wf(wf_P) with the given boolean values as follows.

$$\text{cs}_P([b_1,\ldots,b_n]) = \text{wf}(wf_P)^{b_1,\ldots,b_n}_{v_1,\ldots,v_n}$$

It is possible to add a list of subplans to the wait-for n construct. This is semantically similar to the wait-for n without an additional plan list, but checks only for plans out of this list instead of all subplans.

Note: Although it is not strictly forbidden, be advised, that the use of indirect subplans in the continuation specification is **not** defined. This might or might not work depending on the implementation.

10.4 Open Issues

- ("retry-aborted" in continuation specification) If option "retry aborted" is chosen, aborted sub plans may still complete. This is currently not considered in the continuation specification!

11 Cyclical Plans

Cyclical plans are used to model repetition of a single sub plan. A restricted version of cyclical time annotations is supported here.

11.1 Syntax

cyclical-plan
= cyclical-plan
start-time *cyclical-time-annotation*
name
/ complete condition */*
[*cyclical-complete-condition*]

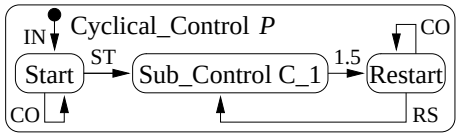
Our simplified cyclical plan consists of a cyclical time annotation which defines a set of starting intervals, the name of a sub plan, and a complete condition.

As complete condition only a single option is supported here.

cyclical-complete-condition
= times-completed *number*

Cyclical time annotations are extensively used in cyclical plans. The difference to time annotations of Sect. 13 is a more complex specification for the reference time point consisting of a time point, an offset and a frequency.

cyclical-time-annotation
= time-range
[starting shift [*minimum*] [*maximum*]]
[finishing shift [*minimum*] [*maximum*]]
[duration [*minimum*] [*maximum*]]
time-point offset frequency



IN: /$i := 1; j := 0$
ST: [$ess + \text{ref}(j) \leq time \leq lss + \text{ref}(j)$]
/ C_1.*consider*
1.5: [**in**(C_1.*Terminated* $\wedge$ $i < n$)]
RS: [$ess + \text{ref}(j) \leq time \leq lss + \text{ref}(j)$]
/ C_1.*retry*; $i := i + 1; j := j + 1$
CO: [$lss + \text{ref}(j) < time$]
/ $j := j + 1$

Figure 16: Controller for cyclical execution

In the following, we will use an abbreviated syntax of cyclical time annotations which is as follows:

[[*ess*, *lss*], [*efs*, *lfs*], [*mindu*, *maxdu*], *tp*, *offset*, *frequency*]

11.2 Semantics overview

For a first version of the semantics we will take a look at the following definition:

cyclical-plan
start-time
[[*ess*, *lss*], [*efs*, *lfs*], [$mindu_1$, $maxdu_1$],
tp, *offset*, *frequency*]
C_1
times-completed *n*

In this definition, plan C_1 is repeated n times.

Slot "start-time" contains a cyclical time annotation. This time annotation defines a set of time points. From this set, the ith time point is used as reference point. The ith time point ref(i) can be calculated according to this formula

$$\text{ref}(i) = tp + offset + (i) \cdot frequency$$

A controller for cyclical plans is as in Fig. 16. As soon as *time* is within the starting interval, sub plan C_1 is considered (transition ST). If C_1 has terminated (transition 1.5) and has not been executed often enough, we will repeat the plan, but will wait for the ith time point in the given cyclical time annotation.

11.3 Open Issues

- Finishing shift [*efs*, *lfs*] and duration [$mindu_1$, $maxdu_1$] of "start-time" are not yet supported.

12 Time Annotations

In the example of Fig. 1, a time annotation has been used to describe the monitoring of conditions over time. This is taken care of in the data abstraction unit (see Sect. 13).

12.1 Syntax

time-annotation
= time-range
[starting-shift [earliest *expression*] [latest *expression*]]
[finishing-shift [earliest *expression*] [latest *expression*]]
[duration
[minimum *expression*] [maximum *expression*]]
reference-point (*expression*|now)*complex-refpoint*

complex-refpoint = (plan-name, plan-state, (enter (l|e)ave))

Within a time-annotation, expressions are used to define a variety of time points. Informally, a plan must be activated within the starting shift. It must complete within the finishing shift and its duration of execution must comply with the minimum and maximum duration. Time values are relative to the given reference point and negative shifts are allowed. In this paper, time annotations are abbreviated as follows

[[*ess*, *lss*], [*efs*, *lfs*], [*mindu*, *maxdu*], *ref*]

and we will use the underscore '_' to represent unspecified values.

[5] describes a number of static checks that a time annotation must satisfy to be considered well-formed. Here, we assume that every time annotation is well-formed.

The referencepoint can be one of three different types. It can be an absolute time, now or a complex-refpoint.

A complex reference point is a link to a (possibly different) plan and a plan state change. An example for such a reference point would be (plan_ob, activated, *enter*).

12.2 Semantics overview

The semantics are already worked into the plan state model as the predicates satisfied and satisfiable. While satisfied is a predicate over medical data (and determines, that a condition has hold long enough and not too long), satisfiable holds until a time out occurs. A time out occurs, once the starting shift has been left and the did not hold at all times since until the finishing shift had been reached. Additionally the minimum and maximum duration must not be violated.

Without the assumption of well formedness of time annotations, the time out condition is much more complex.

Complex reference points are evaluated according to the history of plan events. That is, our previous example, (plan_ob, activated, *enter*), would be a pointer into the time, where plan_ob changed its state to *activated* for the last time. Therefore a complex reference point change its value, after it was first defined. The keywords *enter* and *leave* can be used to refer to the entry or exit of the state. A now reference point can be reduced to an absolute time at the moment of evaluation only, that is, now is the current time. It cannot be statically evaluated.

13 Data Abstraction

Conditions are given as temporal patterns to allow for monitoring of parameters over a longer period of time. Temporal patterns are evaluated in the data abstraction unit.

13.1 Syntax

temporal-pattern
= parameter-proposition
temporal-pattern time-annotation
| simple-condition *formula*
| *temporal-pattern* ($\vee|\wedge$) *temporal-pattern*

A temporal-pattern is either a parameter proposition (including a time annotation), a simple condition or several patterns combined with $\wedge$ or $\vee$. In constrast to parameter propositions, simple conditions are not evaluated over time, as they are first order formulas.

13.2 Semantics overview

The underlying data abstraction unit is not described in detail here and only its purpose is summarized. The semantics of the abstraction unit is not operational, but functional in nature. As input, measurements of patient parameters are taken. The type of parameters can be very different reaching from quantitative values, like bilirubin levels in the blood, to boolean values, e.g. whether the patient is male or female. Data can be provided as a continuous stream of patient readings (high frequency domain as in artificial ventilation of prematured babies) or as sporadic measurements once every month (low frequency, e.g. diabetes mellitus).

The incoming data is memorized in the patient record. Quantitative values can be abstracted to qualitative values. An example has been provided in Sect. 3. More important, the abstraction unit evaluates data over a longer time period, if the data is time annotated in an ASBRU plan. The example

$$\textit{bilirubin_decrease} < 1\ [[-2\ \text{h}, _], [_, 0\text{h}], [_, _], \textit{now}]$$

requires monitoring the decrease of bilirubin level over a period of the last 2 hours.

As output of the abstraction unit, the truth values of conditions are provided. As evaluation of a condition may take time, the result is either *true*, *false*, or yet *unknown*. A condition is *false* only, if it cannot be satisfied in the future. Otherwise, it would be considered *unknown*. To prevent a three valued logic, this is done by two predicates, satisfied and satisfiable. A condition is satisfiable, if there is a continuation of the patient data and (a possibly different) condition, such that this condition is satisfied in the future.

One should keep in mind, that there are conditions, that are technically satisfiable, while it can be decided, that this condition could never be medically satisfied. As an example consider a condition, postulating that for the next five hours at least ten minutes there is high blood pressure. A dead person could never medically fullfil this condition, while it is still technically possible, as ten minutes is less than five hours.

To understand the predicate satisfied, it is crucial to distinguish whether the time annotated condition evaluates to true or the underlying (possibly also time annotated) condition evaluates true. The time annotated condition evaluates to true if and only if there is an interval *I* during which the underlying condition evaluated to true. Eight criteria have to be met for this interval:

- *I* is at least as long as the minimum duration
- *I* is not longer than the maximum duration
- the starting point of *I* is not sooner than the reference point plus the earliest starting shift.
- the starting point of *I* is not later than the reference point plus the latest starting shift.
- the finishing point of *I* is not sooner than the reference point plus the earliest finishing shift.
- the finishing point of *I* is not later than the reference point plus the latest finishing shift.
- the underlying condition was not true directly before the earliest time of *I* or the earliest time of *I* is the earliest starting shift plus the reference point.
- the underlying condition was not true directly after the latest time of *I* or the latest time of *I* is the latest finishing shift plus the reference point.

For a condition to be satisfiable, any of the following three properties must hold:

- the condition is already satisfied
- the latest starting point has not been reached
- the condition can still fulfill the time annotation

Be t_1 the starting point of interval I, t_2 the finishing point of I, ta = [[*ess*, *lss*], [*efs*, *lfs*], [*minDu*, *maxDu*], *refPoint*] the time annotation of a temporal pattern and c the corresponding (underlying) temporal pattern.

satistfied(c, ta, I)
$\leftrightarrow$ I is normalized
$\wedge\ t_1 \geq$ *ta.ess* + *ta.refPoint*
$\wedge\ t_1 \leq$ *ta.lss* + *ta.refPoint*
$\wedge\ t_2 \geq$ *ta.efs* + *ta.refPoint*
$\wedge\ t_2 \leq$ *ta.lfs* + *ta.refPoint*
$\wedge\ t_2 - t_1 \leq$ *ta.maxDu*
$\wedge\ t_2 - t_1 \geq$ *ta.minDu*
$\wedge$ satisfied(c, I)
$\wedge\quad \neg \exists\, t_3.\ t_2 < t_3,$
$\wedge$ satisfied(c, $t_1 \times t_3$)
$\vee$ *ta.lfs* = $\infty \wedge$ *ta.maxDu* = ∞
$\wedge\quad \neg \exists\, t_0.\ t_0 < t_1,$
$\wedge$ satisfied(c, $t_0 \times t_2$)
$\vee$ *ta.ess* = $-\infty \wedge$ *ta.maxDu* = ∞

satistfiable(c, ta, I)
$\leftrightarrow$ I is normalized
$\wedge\quad t_2 <$ *lss* + *refPoint*
$\wedge$ satisfied(c, t_2)
$\vee\ t_2 <$ *lss* + *refPoint* - 1
$\vee\ t_2 <$ *ess* + *refPoint*
$\vee\ t_2 <$ *lfs* + *refPoint*
$\wedge\ t_1 \geq$ *ess* + *refPoint* $\wedge\ t_1 \leq$ *lss* + *refPoint*
$\wedge$ satisfied(c, I)
$\wedge \neg$ satisfied(c, $t_1 - 1$)
$\wedge\ t_2 - t_1 \leq$ *maxDu*
$\vee$ satisfied(c, ta, I)

For the sake of completeness, the definitions for the other cases of the temporal pattern are also given here. Be tp_1, tp_2 temporal patterns, sc a simple condition with formula φ.

$$\text{satisfied}(tp_1 \wedge\ tp_2, I) \leftrightarrow \text{satisfied}(tp_1, I) \wedge \text{satisfied}(tp_2, I)$$
$$\text{satisfied}(tp_1 \vee\ tp_2, I) \leftrightarrow \text{satisfied}(tp_1, I) \vee \text{satisfied}(tp_2, I)$$
$$\text{satisfied}(sc, I) \leftrightarrow \forall\ t.t \in\ I \rightarrow\ \varphi(\sigma(t))$$

where σ is the global state.

14 Effects

Effects are a method of expressing the outcome of a plan. This outcome can be wanted or unwanted side effects, the administration of a treatment may have.

14.1 Syntax

The syntax of a effect is as follows.

effect = *effect_formula*
earliest starting point
latest starting point
earliest finishing point
latest finishing point

An effect consists of one first order formula and four complex reference points, i.e. a reference-point as in Chapter 12.

14.2 Semantics

The first order logic formula describes the influence of the treatment on the patient. It can be defined totally (i.e. a decrease of five units per second), partially (i.e. an increase of five to seven units per second) or left underspecified (i.e. the value decreases).

The influence of the effect starts indeterministically sometime within the starting shift and ends indeterministically within the finishing shift. The reference points that define the starting shift and the finishing shift are complex time points like in Chapter 12. Additionally they allow for basic mathematics, that is addition and subtraction of constants. Therefore, it can be stated, that the effect startes 5 hours after activation of plan ”treatment-radiotherapy” by using the reference point

leave(ready, treatment-radiotherapy) + 5h

The time span in which the effect will take place is indeterministic. It is only guaranteed, that it will hold from the latest starting point onwards to the earliest finishing point. If the starting and the finishing interval do overlap, it is not guaranteed, that the effect will occur at all.

15 Intentions

Intentions describe temporal properties of plans and can be verified as described next.

15.1 Syntax and semantics overview

intentions
= intentions
((intermediate-state|overall-state)
(avoid|maintain|achieve)
temporal-pattern)*

An intention is to either avoid, maintain, or achieve an overall or intermediate state which is described by a temporal pattern.

Intentions can be translated into temporal logic. Details are omitted here.

15.2 Verification

One major goal of our project is to formally verify the operational behaviour of Asbru plans against properties which are expressed as intentions. For the example in Sect. 3 the task would be to verify that *bilirubin* is never equal to transfusion throughout execution – which should be easy – and if the plan completes, *bilirubin* equals to Observation – which is not so obvious.

For verification we are using the interactive theorem prover KIV. We regard automatic verification not powerful enough to deal with the data involved and therefore an interactive verifier is necessary. However it would be worthwhile to define sub tasks which could be treated with model checkers. KIV already supports the verification of parallel programs against properties expressed in temporal logic. The verification strategy is to symbolically execute programs and to use induction, if necessary [2]. A similar approach shall be applied to Asbru plans.

Because of its functional nature, the tasks of the data abstraction unit can be translated directly into algebraic specifications. The difficulty is to capture the operational, state based, parallel behaviour of the Asbru plans themselves. Encoding the SOS rules representing the formal semantics directly into proof rules has been tried but turned out to be too inefficient. Hundreds of proof steps were necessary to execute one Asbru step. In part, this is because of the explicit encoding of the plan hierarchy. A more direct representation of Asbru plans with higher level proof rules is necessary. Currently we are translating Asbru plans into parallel programs preserving the hierarchy of parent and sub plans. With this representation we are able to verify the example intentions, which are translated into temporal logic. However, this translation is only possible for a subset of features and its correctness needs to be examined still. In a future step we would like to directly support Asbru syntax and design proof rules for executing Asbru. In order to be correct, these proof rules still need to adhere to the formal semantics presented here.

16 Conclusion

The formal semantics of major concepts of Asbru has been explained in this paper. We are confident that the formal semantics alone will help to better understand Asbru plans and thereby improve quality of medical protocols. Furthermore the semantics is an important link between the modelling language and the representation in KIV. An overview on how we will use KIV to verify properties formally has been given. Further research on this topic will be our next major step.

Initially, the formal semantics of Asbru has been given in the form of SOS rules. However, these rules turned out to be difficult to understand. Representing rules as transitions in statecharts resulted in a more compact and intuitive picture of plan behaviour. Even if some of the technical details of the semantics are not correctly captured within these graphics, the statecharts are very suitable for discussions and language documentation.

Acknowledgements

This work was possible only thanks to long and fruitful discussions with Silvia Miksch (TU Vienna) and her group Andreas Seyfang and Robert Kosara. Also many thanks to Frank van Harmelen, Mar Marcos (VU Amsterdam), and Annette ten Teije (Univ. Utrecht) for their assistance and good collaboration. This work has been partially funded by the European Commission's IST program, under contract number IST-FP6-508794 Protocure II.

References

[1] M. Balser, C. Duelli, and W. Reif. Formal semantics of Asbru – An Overview. In *Proceedings of IDPT 2002*. Society for Design and Process Science, 2002.

[2] M. Balser, C. Duelli, W. Reif, and G. Schellhorn. Verifying concurrent systems with symbolic execution. *Journal of Logic and Computation*, 12(4):549–560, 2002.

[3] J. Bury, J. Fox, and D. Sutton. The PROforma guideline specification language: progress and prospects. In *Proceedings of the First European Workshop, Computer-based Support for Clinical Guidelines and Protocols (EWGLP 2000)*, 2000.

[4] W. Damm, B. Josko, H. Hungar, and A. Pnueli. A compositional real-time semantics of STATEMATE designs. In W.-P. de Roever, H. Langmaack, and A. Pnueli, editors, *COMPOS' 97*, volume 1536 of *LNCS*, pages 186–238. Springer, 1998.

[5] G. Duftschmid and S. Miksch. Knowledge-based verification of clinical guidelines by detection of anomalies. *Artificial Intelligence in Medicine*, Special Issue: Workflow Management and Clinical Guidelines in Medicine(22(1)), 2001.

[6] S. Herbert, C. Gordon, A. Jackson-Smale, and S. Renaud. Protocols for clinical care. *Copmputer Methods and Programs in Biomedicine*, 48, 1995.

[7] S. Miksch, Y. Shahar, and P. Johnson. Asbru: A task-specific, intention-based, and time-oriented language for representing skeletal plans. In E. Motta, F. v. Harmelen, C. Pierret-Golbreich, I. Filby, and N. Wijngaards, editors, *7th Workshop on Knowledge Engineering: Methods & Languages (KEML-97)*. Milton Keynes, UK, 1997.

[8] L. Ohno-Machado, J. Gennari, S. Murphy, N. Jain, S. Tu, D. Oliver, E. Pattison-Gordon, R. Greenes, E. Shortliffe, and G. Barnett. The GuideLine Interchange Format: A model for representing guidelines. *American Medical Association*, 5, 1998.

[9] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In *Symposium on Theoretical Aspects of Computer Software*, volume 526 of *LNCS*, pages 244–264. Springer, 1991.

[10] Protocure – Improving medical protocols by formal methods. `http://www.protocure.org`.

[11] A. Seyfang, R. Kosara, and S. Miksch. Asbru's reference manual, asbru version 7.2, document revision 1. Technical report, Vienna University of Technology, Institute of Software Technology, 2000.